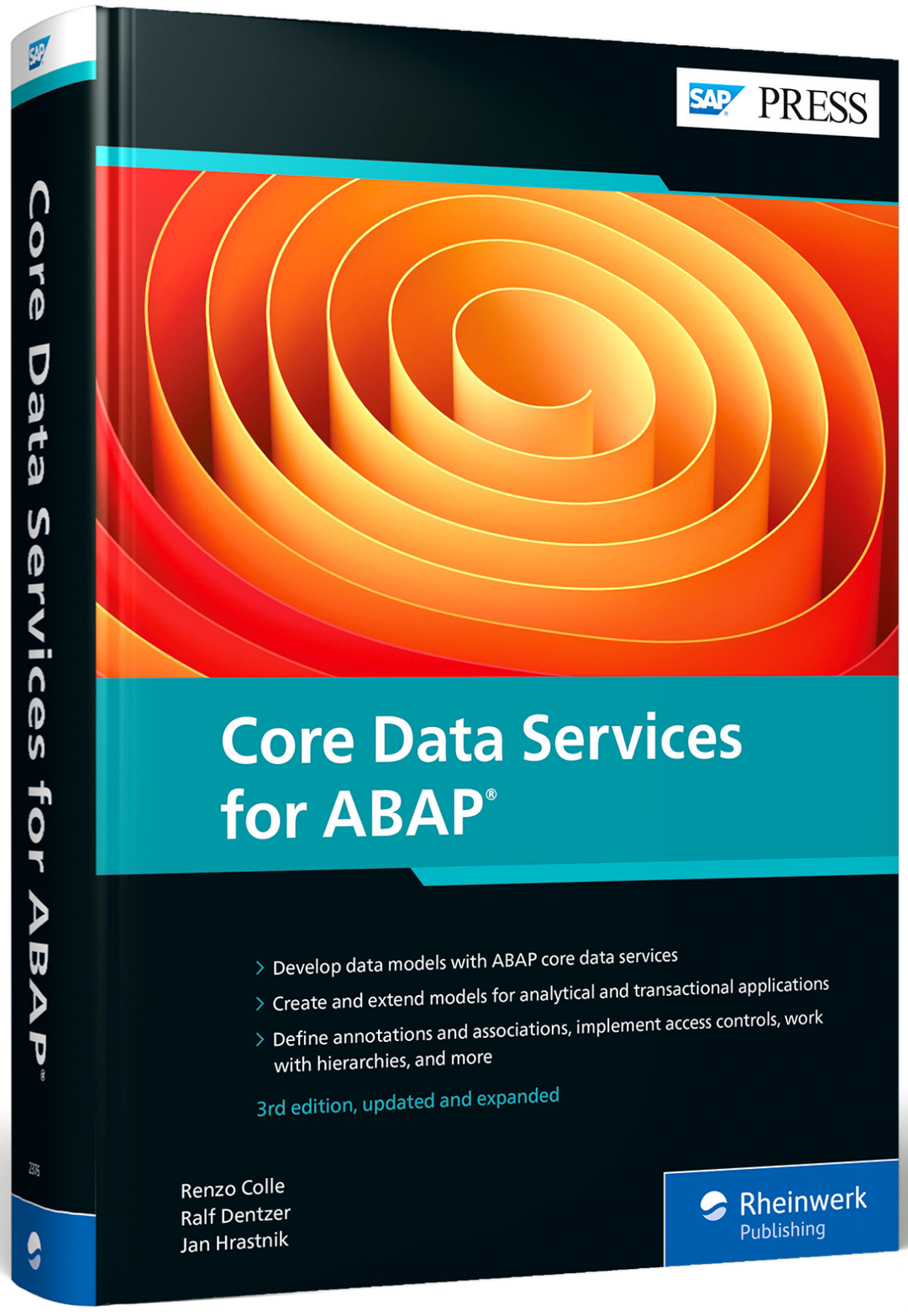


Reading Sample

In this chapter, you'll learn about the technical basics of CDS data models. Walk through syntax elements such as key fields, cast operations, simple types, projection fields, and more. Then, familiarize yourself with associations and annotations.

Colle, Dentzer, Hrastnik

Core Data Services for ABAP

754 pages | 10/2023 | $89.95 | ISBN 978-1-4932-2376-3

www.sap-press.com/5642

Chapter 2

Fundamentals of CDS Data Modeling

This chapter explains the basic syntax and concepts of core data services (CDS) for defining data models. It focuses on the most important design-time artifact—the CDS view—and describes its components.

Important terms

CDS allows you to define and implement data models. The implementation of these data models is based on the CDS *data definition language* (DDL), which is closely related to the *structured query language* (SQL). It supports defining several specialized *CDS entity types* and related metadata models.

Structure of this chapter

In this chapter, Section 2.1 provides an overview of these models and their purposes. The remainder of the chapter deals with CDS view models, which, from a developer's perspective, are considered the most important CDS entities. Specifically, you'll learn how to implement your *data selections* using CDS syntax. You'll find an overview of the fundamental syntax elements for CDS view definitions in Section 2.2. Afterward, we'll introduce you to the following aspects in more detail:

- Key fields (Section 2.3)
- Cast operations (Section 2.4)
- Typed literals (Section 2.5)
- Simple types (Section 2.6)
- Case statements (Section 2.7)
- Session variables (Section 2.8)
- Client handling (Section 2.9)
- Select distinct statements (Section 2.10)
- Union views (Section 2.11)
- Intersect and except statements (Section 2.12)
- Joins (Section 2.13)
- SQL aggregation functions (Section 2.14)
- Projection fields (Section 2.15)
- Parameters (Section 2.16)
- Reference fields (Section 2.17)
- Conversion functions for currencies and quantity units (Section 2.18)

- Provider contracts (Section 2.19)
- Entity buffer definitions (Section 2.20)

Enrichments of the data selection logic

Apart from modeling data selections, you'll also get acquainted with CDS associations and annotations. *Associations* allow you to establish directed relationships between your data models. These associations enrich the data models with semantic information. Furthermore, they can be leveraged when defining `select` statements. Associations are described in detail in Chapter 3.

Annotations allow you to equip your CDS models with additional metadata that is primarily expected to be interpreted by the consumers of the CDS models. For example, CDS annotations are evaluated by the Service Adaptation Definition Language (SADL) infrastructure when defining an OData service. You'll learn how annotations are defined and applied in Chapter 4.

2.1 Overview of CDS Models

CDS models comprise the definitions of various CDS entity types such as CDS views and CDS abstract entities. These entity types will cover different use cases. Typically, the CDS entity type can't be changed after the corresponding CDS model is activated. From a technical perspective, all CDS entity types are assigned to the object type `DDLS`, which we introduced in Chapter 1, Section 1.2.2. However, the range of CDS models also comprises further objects such as CDS metadata extensions and simple types, which have deviating object types of `DDLX` and `DRTY`, respectively.

CDS model definitions

Table 2.1 shows the fundamental definitions of the available CDS models along with their envisioned usages.

CDS Model	Definition	Application
View	`define view ... as select from ...`	Data selection
View entity	`define view entity ... as select from ...`	Data selection; successor of view CDS model
Projection view	`define view entity ... as projection on ...`	Data selection
Transient view	`define transient view entity ... as projection on ...`	Analytical query

Table 2.1 CDS Models

CDS Model	Definition	Application
Extend view	`extend view ... with ...`	Extension of a CDS view
Extend view entity	`extend view entity ... with ...`	Extension of a CDS view entity
Table function	`define table function ...`	Data selection with SAP HANA–native functions
Custom entity	`define custom entity ...`	ABAP-based data selection in SADL-based OData services
Extend custom entity	`extend custom entity…with…`	Extension of a CDS custom entity
Abstract entity	`define abstract entity ...`	Modeling of structures
Extend abstract entity	`extend abstract entity…with…`	Extension of an abstract entity
Hierarchy	`define hierarchy entity ...`	Modeling of hierarchies
Metadata extension	`annotate view ... with ...`	Annotation decorator
Simple type	`define type …`	Typing of fields and parameters; successor of ABAP Data Dictionary data element in CDS
Entity buffer definition	`define view entity buffer on ...`	Buffering records of CDS view entities

Table 2.1 CDS Models (Cont.)

As shown, there are different types of CDS view models:

Views

- **Views (aka V1 views)**
  Represent the original CDS view models.
- **View entities (aka V2 views)**
  Represent the successor of V1 views. Compared to V1 views, V2 views don't generate an additional ABAP Data Dictionary view upon their activation. This reduces the risk of technical inconsistencies and improves the overall activation performance. Furthermore, V2 views enforce more

homogenous modeling and apply stricter syntax checks. For example, V2 views foster uniform client handling, which avoids some of the issues that might occur when defining and using V1 views. Because primarily view entities are expected to benefit from further improvements of the CDS infrastructure, we recommend always defining V2 views (probably as projection or transient views) instead of V1 views.

Migration of V1 Views to V2 Views

You can migrate your existing V1 views to V2 views. This migration can be achieved by manually changing the statement `define view …` to `define view` **`entity`** …. Alternatively, you can run report RUTDDLSV2MIGRATION or use the corresponding functionality **Migrate to CDS View Entity** of the context menu in the **Project Explorer** view of the ADT environment.

However, before migrating a V1 view, all the usages of its generated SQL view need to be removed. Additionally, the definitions of the V1 views and their extensions need to be aligned with the stricter syntax checks and behavior of V2 views prior to their migration. The migration tools just mentioned perform some of the necessary adjustments automatically and provide you with information about potential changes. Therefore, we suggest using the tools for migrating V1 views.

- **Projection views**
  Represent a specialization of the view entities. Their main purpose is the definition of interfaces on their underlying CDS models with a modeled mapping of the corresponding functionality. As a result, projection views restrict the overall functionality of view entities to mere projection features. Projection views will be explained later in Chapter 6, Section 6.1.
- **Transient views**
  Transient views (sometimes also referred to as V3 views) define CDS view entities without a direct representation on the SAP HANA database system. They act as mere declarative view models whose runtime behavior is governed and implemented by infrastructure components such as the analytical engine. This implies that you can neither use transient views as data sources for other CDS views nor can you select from transient views in your ABAP code. You'll learn about the use of transient views in the context of implementing analytical queries in Chapter 10, Section 10.3.

For the sake of brevity, we'll refer to *CDS view entities* when talking about CDS views or simply *views* unless explicitly mentioned otherwise.

View extensions

Extend views and *extend view entities* allow you to enhance a CDS view or a CDS view entity, respectively, with additional fields and associations; that is, they allow you to define extensions of CDS data models. You'll learn more about such extensions in Chapter 15.

Table functions

Whereas the selection logic of CDS views is implemented in the definitions of the CDS models themselves by applying a declarative CDS syntax, the implementation of *table functions* leverages native SAP HANA SQLScript syntax. The usage of SAP HANA SQLScript enables a higher degree of freedom in the way the selection logic is implemented. In addition, it provides you with the option to use functions that aren't yet supported by the CDS syntax. However, there are also several drawbacks to using table functions. You'll find further information about CDS table functions in Chapter 7.

Custom entities

Similar to table functions, *custom entities* only capture the signature of a data model within the CDS model definition itself. The actual implementation of custom entities occurs in ABAP code. As such, the logic of custom entities can't be executed on the database level. Instead, custom entities can be used to define OData entity sets that are processed by the SADL infrastructure. For more information, refer to *http://s-prs.co/v529401*.

ABAP Logic Can't Be Used in Logic Accessed from SAP HANA

You can't incorporate ABAP logic into the logic being executed on SAP HANA. ABAP logic can only be processed before or after processing the logic on SAP HANA.

Abstract entities

Abstract entities are used to define signatures without implementation. In the context of the *ABAP RESTful application programming model*, you can use them for defining data structures for parameters and results of actions and functions, as well as for event payloads. Furthermore, abstract entities may be used for defining proxies of external service models entities. For more information, refer to the SAP documentation at *http://s-prs.co/v529421*.

Hierarchies

Hierarchy entities allow you to gain access to the functionality of SAP HANA hierarchies; that is, they allow you to leverage SAP HANA hierarchy features in your ABAP code. You'll find an explanation of the hierarchy modeling and its application in Chapter 12, Section 12.3.

Metadata extensions

Whereas all the aforementioned CDS models represent dedicated CDS entity types, *metadata extensions* represent an extension option of the CDS entity type instead. From a technical perspective, they are assigned the object type `DDLX`. The purpose of metadata extensions is to enrich and redefine annotations of CDS entities. The concept of metadata extensions will be explained in more detail in Chapter 4, Section 4.4.

CDS simple types

CDS simple types define reusable scalar type definitions. From the CDS perspective, simple types can be interpreted as successors of the ABAP Data Dictionary data elements. We'll demonstrate possible definitions and usages of simple types in Section 2.6.

Entity buffer definitions

For CDS view models with simple logic, a buffer can be defined on the application server. In this context, *entity buffer definitions* capture the required administrative data similar to table buffers that are managed by the ABAP Dictionary. Direct data selections from buffered CDS views via the ABAP SQL interface are automatically transferred to the buffer by the ABAP runtime environment, if the request can be processed on the buffer. This can help reduce the load of the SAP HANA database and speed up the provisioning of the requested data. You'll find further information about entity buffer definitions in Section 2.20.

2.2 Overview of CDS View Syntax

Example: CDS view definition

CDS views define `select` statements, which are augmented with additional metadata information. Listing 2.1 provides an example of such a definition.

```
/*comment*/
//other comment
//annotations
@AccessControl.authorizationCheck: #MANDATORY
@EndUserText.label: 'View Definition'
//view definition
define view entity Z_ViewDefinition
  //parameter definition
  with parameters
    P_SalesOrderType : auart
  //data source of selection with alias name
  as select from ZI_SalesOrderItem as ITEM
  //join
    left outer to exact one join ZI_SalesOrder as SO
      on SO.SalesOrder = ITEM.SalesOrder
  //association definition
  association [0..1] to ZI_Product as _Product on
    $projection.RenamedProduct = _Product.Product
{
      //projected field as key
  key ITEM.SalesOrder,
      //projected field used in association definition
  key ITEM.Product                        as RenamedProduct,
```



```
      //constant
      abap.char'A'                        as Constant,
      //calculated field
      concat( ITEM.SalesOrder, ITEM.Product ) as CalculatedField,
      //aggregate
      count(*)                            as NumberOfAggregatedItems,
      //projected association
      ITEM._SalesOrder,
      //association exposure
      _Product
}
//filter conditions based on join partner and parameter
where
  SO.SalesOrderType = $parameters.P_SalesOrderType
//aggregation level
group by
  ITEM.SalesOrder,
  ITEM.Product
```

Listing 2.1 CDS View Definition

Example analysis

Let's now take a closer look at this definition:

- **`define` statement**
  The `define` statement specifies the technical type of the CDS model. In the given example, a CDS view entity model named `Z_ViewDefinition` is defined (`define view entity`).
- **Data sources**
  CDS view `Z_ViewDefinition` uses the CDS view for sales order items (`ZI_SalesOrderItem`) as its primary data source (`select from`). It combines this primary data source via a `left outer join` statement with a secondary data source, which provides the header data of sales orders `ZI_SalesOrder`. Both data sources are locally renamed by alias names `ITEM` and `SO`, respectively. You'll learn more about join types in Section 2.13.
- **Projected elements**
  CDS view `Z_ViewDefinition` projects various elements from its primary data source, for instance, field `SalesOrder`, which identifies the sales order, and field `Product`, which establishes a relationship between the sales order item and its assigned product. The latter field is renamed by aliasing it in `RenamedProduct`.
- **Aggregation**
  Due to the applied aggregation logic of the data records (`group by`) by

fields `SalesOrder` and `Product`, both of these fields determine the `key` of CDS view `Z_ViewDefinition`.

- **Calculated fields**
  Besides the projected fields, CDS view `Z_ViewDefinition` also defines the field `Constant` locally, which has the value `A` for all records. Technically, this field is defined as a typed literal (Section 2.5), which is based on the elementary ABAP Data Dictionary type `char` of length `1`. Furthermore, it contains field `CalculatedField`, which concatenates the values of fields `SalesOrder` and `Product` by applying function `concat`.

  Field `NumberOfAggregatedItems` represents an aggregate. Its value is determined by counting (`count(*)`) the records of sales order items, which are grouped in a single record of the selection result.
- **`where` condition**
  The data selection is filtered by a `where` clause, which compares the sales order type (`SalesOrderType`) with the value of CDS parameter `P_SalesOrderType`.
- **Association**
  The CDS view model also defines association `_Product`, which establishes a named relationship to the corresponding record of the CDS view of product header `ZI_Product`. The association is exposed by incorporating it in the projection list of the view. As a result, the association can be used by consumers of the CDS view too.
- **Comments**
  Within the CDS view model, *comments* refer to single-line comments prefixed with double slashes (//) or incorporated into the pattern /*…*/, which allows you to define multiline comments.
- **Annotations**
  Annotation `@AccessControl.authorizationCheck: #MANDATORY` indicates that the selection result will be subject to an access control. You'll learn more about how to enable such authorization checks in Chapter 5. Annotation `@EndUserText.label` specifies a language-dependent text for the view that will correspond to its description.

Detailed information

As you can see from the presented example, the CDS syntax consists of various different elements. To master the definition of CDS models yourself, we'll provide you with more detailed information about these syntax elements in the subsequent sections and chapters of this book.

Technical aspects

The definition of a CDS view model is captured as source code in the transportable object of type `DDLS`. The CDS view isn't transported; instead, it's

generated locally when activating the `DDLS` object. In the ABAP repository, it has object type `STOB` (CDS view). For CDS view entities, the name of the CDS view (converted to uppercase letters) has to match the name of its defining `DDLS` object.

> **Consider the Namespace of CDS Models**
>
> The CDS view name shares the same namespace with other ABAP Data Dictionary artifacts (e.g., tables and views). This restriction has to be considered when choosing names for CDS views and `DDLS` objects.
>
> In addition, to avoid name clashes with SAP-delivered artifacts, you should always use your dedicated customer or partner namespace when defining your own CDS models.

2.3 Key Fields

Key definition

CDS view fields are defined as *key fields* by adding the preceding syntax element `key` to the fields. In general, the key of a CDS model can be composed of multiple key fields. These key fields must be placed before the non-key fields in the projection list of the CDS view. The key must be defined in such a way that it uniquely identifies a single record in the results list of a data selection. It's recommended to keep the key as short as possible; that is, fields should only be included in the key definition if they are required for achieving an unambiguous identification of a single record in the results list of a data selection.

> **Client Field**
>
> CDS view entities apply an automated filtering based on the client (Section 2.9). Therefore, the client field isn't contained in the list of key fields, even if the underlying database records depend on the client.

Example: CDS view with key fields

The sample CDS view for sales order item `ZI_SalesOrderItem` in Listing 2.2 has a key that consists of the two fields `SalesOrder` and `SalesOrderItem`. Both fields are required key components because the identifier of the sales order item is only uniquely defined in the context of its embedding sales order document. From a technical perspective, it's also possible to mark field `Product` as another key field. However, this field isn't required for making the key of the CDS view unique, so it isn't included in the illustrated key definition.

```
define view entity ZI_SalesOrderItem
  as select from zsalesorderitem
{
  key salesorder     as SalesOrder,
  key salesorderitem as SalesOrderItem,
      product        as Product
}
```

Listing 2.2 CDS View with Key Definition

Using the key definition

The key definition serves as documentation of a CDS view model. In addition, it can be used for consistency checks. For example, you can check the specified cardinality of an association based on the bound key fields of the target CDS model. Key definitions may also have an impact on the CDS access controls that evaluate the key definitions when injecting authorization restrictions into `select` statements. Furthermore, it's evaluated by various implementation frameworks, which provide their services on top of the corresponding CDS view. For instance, the key definition of a CDS model can be used for automatically deriving the key definition of an *OData entity set*, which is mapped onto the CDS model. Chapter 6 provides you with more information about the OData exposure of CDS models.

Increase Transparency of CDS Models

You should always define a key for your CDS models if feasible from a technical perspective. A key definition should only be omitted if the CDS model doesn't contain suitable fields or combinations for defining a unique key.

2.4 Cast Operations

Cast operations

Cast operations represent some of the most important fundamental SQL functions. You can use cast operations for determining the type of a calculated field and for converting the type of existing fields on the database level. The CDS syntax supports casts onto elementary ABAP types as well as onto data elements and simple types. By using data elements or simple types, you not only can change the type of the cast fields but also assign suitable annotations and properties such as label texts and conversion exits to them.

Cast operations without type changes

If a cast doesn't change the technical type of the field but, for example, is performed to exchange its label texts, you can enrich the cast statement with addition `preserving type`. This addition informs the compiler that no

effective type change is required, and, as a result, no cast is required on the database level.

Supported Type Conversions

Not all type conversions are supported by the CDS syntax. You'll find an overview of the supported conversions in your ABAP documentation, for example, at *http://s-prs.co/v529402*.

Some type conversions aren't supported by a cast operation and can only be achieved by applying dedicated CDS conversion functions. For example, you can only convert a floating point value to a decimal value by using function `fltp_to_dec`.

In a CDS table function implementation, the extended options of native SAP HANA conversion functions are available. Thus, engaging a CDS table function may allow you to perform conversions that aren't supported by the CDS syntax itself.

Example: CDS view with type conversions

Listing 2.3 shows a CDS view with fields that have different type conversions.

```
define view entity Z_ViewWithCasts as select distinct from t000
{
  t000.logsys                          as ProjectedField,
  '20170809'                           as CharacterField,
  cast ( '20170809' as abap.dats )     as DateField,
  cast ( cast ( 'E' as abap.lang ) as sylangu preserving type )
    as LanguageField,
    1.2 as FloatingPointField,
  fltp_to_dec( 1.2 as abap.dec(4,2) ) as DecimalField
}
```

Listing 2.3 CDS View with Cast Operations

Example analysis

By default, a field keeps the type of its origin if it's simply projected into a CDS view. For example, field `ProjectedField` retains the type from its underlying base field `t000.logsys`.

If fields are locally defined, an implicit type assignment is applied. As a result, field `CharacterField` is defined as a `char` field of length `8` based on the literal value `20170809`. By explicitly casting the same literal value onto type `dats`, field `DateField` becomes a date field.

You can also nest cast operations. This option is illustrated by expression `cast ( cast ( 'E' as abap.lang ) as sylangu preserving type )`. The value `E` is implicitly regarded as a character value of length `1`. In a first step, the type

of corresponding field `LanguageField` is explicitly set to `lang`. In a second step, data element `sylangu` finally determines the type of the field. Because there is no change of the technical type, this conversion is accompanied by addition `preserving type`.

By assigning value `1.2` to field `FloatingPoint`, this field is implicitly typed by `fltp`. In addition, field `DecimalField` is constructed from value `1.2`. Due to this value assignment, it's implicitly defined as a field of type `fltp` too. To convert this field into a field of type `dec`, function `fltp_to_dec` is applied.

[+]

Use Explicit Type Assignment

You should always explicitly cast calculated fields onto the desired data type if feasible from a technical perspective. Otherwise, you may experience unexpected side effects from implicit type assignments.

If you're not sure about the type of a CDS field, you can position your cursor on the CDS view or one of its fields and press F2. This launches an information popup, as shown in Figure 2.1, where you can find the previously discussed type information.

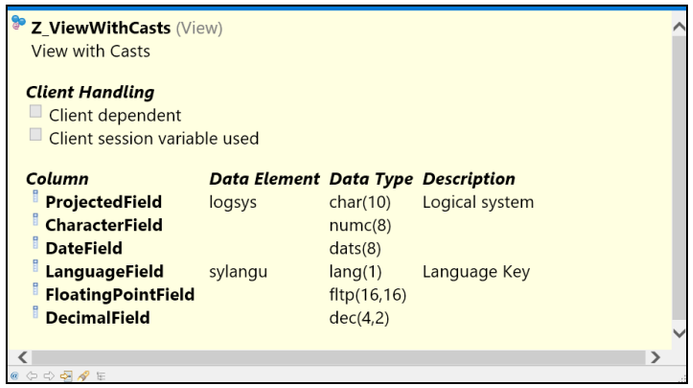


Figure 2.1 Technical Field Information of the CDS View from Listing 2.3

2.5 Typed Literals

Typed literals

Typed literals allow you to specify the technical ABAP type of a constant value, which you introduce in your CDS model. To make a plain literal become a typed literal, you have to enclose the actual value in single quotation marks and add the type information as a prefix. Listing 2.4 shows some examples.

Example: CDS view with typed literals

```
define view entity Z_ViewWithTypedLiterals
  as select distinct from t000
{
  '  Char10  '                          as CharacterValue,
  cast( '  Char10  ' as abap.char(10) ) as CastCharacterValue,
  abap.char'  Char10  '                 as TypedCharacterValue,
  cast( abap.char'  Char10  '           as Char10 preserving type )
                                        as CastTypedCharacterValue,
  1234.56                               as FloatingPointValue,
  abap.fltp'1234.56'                    as TypedFloatingPointValue,
  fltp_to_dec(1234.56 as abap.dec(6,2)) as ConvertedDecimalValue,
  abap.dec'1234.56'                     as TypedDecimalValue,
  abap.dec'001234.5600'                 as TypedDecimalValue2
}
```

Listing 2.4 CDS View with Typed Literals

Example analysis

In this example, field `CharacterValue` is derived from literal `'  Char10  '`. From the CDS perspective, this field is implicitly typed as a character field of length 8; that is, the leading two spaces are considered, whereas the last two spaces are ignored.

Character like values

With an explicit cast, you can enforce the field to have a length of 10 characters. This is shown by field `CastCharacterValue`. Similarly, field `TypedCharacterValue`, which is derived from the typed literal `abap.char'  Char10  '`, becomes a character field of length 10 automatically. In other words, for typed literals, the entire value entered in the quotation marks is considered to be significant.

If you define a field based on a typed literal and want to equip it with additional metadata such as label texts, you should wrap the typed literal with an explicit type-preserving cast onto a suitable data element or simple type. This is illustrated by field `CastTypedCharacterValue`.

Non-character like values

Typed literals become especially important if the literals don't represent mere character strings. This will be demonstrated by the remaining fields that are defined in Listing 2.4.

When defining a value such as `1234.56`, it's implicitly interpreted as a floating point value. Consequently, `FloatingPointValue` and `TypedFloatingPointValue` share the same technical typing. If you want to define a decimal value instead, you basically have two options: (1) perform an explicit type conversion of such a floating point value by applying function `fltp_to_dec`, as illustrated for field `ConvertedDecimalValue`; or (2) define a typed literal. Field `TypedDecimalValue` in Listing 2.4 is typed accordingly. However, note that `abap.dec'1234.56'` is interpreted as a decimal value from the very

beginning. In contrast, floating point value `1234.56` has to be effectively converted to a decimal value. This not only implies that there is some overhead in the processing logic of the value but also may be the reason the field values returned by a view differ. In our case, `TypedDecimalValue` exposes modeled value `'1234.56'`, whereas `ConvertedDecimalValue` returns converted value `'1234.55'`.

As mentioned before, the value and its notation are specified within the typed literal matter. Typed literal `abap.dec'1234.56'` is associated with a decimal type of length `6` with `2` decimals. Typed literal `abap.dec'001234.5600'` yields from typing field `TypedDecimalValue2` as a decimal field of length `10` with `4` decimals.

Use Typed Literals

We recommend always using typed literals instead of plain literals. They help you avoid making errors and introducing unnecessary conversions.

Influence type length

As already explained the concrete value defined in a typed literal is decisive. For example, typed literal `abap.dec'1234.56'` results in field `TypedDecimalValue` having type `dec` of length `6` with `2` decimals. In contrast, typed literal `abap.dec'001234.5600'` results in field `TypedDecimalValue2` being defined as a decimal field with length `10` and `4` decimals.

2.6 Simple Types

Simple type definitions

CDS simple types can be specified based on different type definitions: Listing 2.5 illustrates simple type `ZBT_LanguageA`, which is based on elementary dictionary type `lang`.

```
define type ZBT_LanguageA : abap.lang;
```

Listing 2.5 Simple Type Based on an Elementary ABAP Data Dictionary Type

Listing 2.6 shows simple type `ZBT_LanguageB`, which is based on data element `spras`.

```
define type ZBT_LanguageB : spras;
```

Listing 2.6 Simple Type Based on a Data Element

Listing 2.7 demonstrates the definition of simple type `ZBT_LanguageC`, which is based on the simple type `ZBT_LanguageB` from Listing 2.6.


```
@EndUserText.label: 'Language Type C'
@Semantics.language: true
define type ZBT_LanguageC : ZBT_LanguageB;
```

Listing 2.7 Simple Type Based on Another Simple Type

Enrichment with annotations

In addition, Listing 2.7 shows how you can enrich the mere technical definition of a simple type with annotations. In the given case, simple type `ZBT_LanguageC` is classified as a language code by its annotation `@Semantics.language: true`. In addition, it receives the label text `'Language Type C'` from its annotation `@EndUserText.label`.

Multilevel definitions of simple types

You can define multilevel hierarchies of simple types, as shown in Listing 2.8. Therein, simple type `ZBT_LanguageD` is typed by simple type `ZBT_LanguageC` from Listing 2.7.

```
@EndUserText.label: 'Language Type D'
@ObjectModel.sapObjectNodeTypeReference: 'Language'
define type ZBT_LanguageD : ZBT_LanguageC;
```

Listing 2.8 Simple Type Based on Another Simple Type: Multilevel

In this context, the annotations of simple type `ZBT_LanguageC` are propagated to simple type `ZBT_Language` such as the technical properties of the underlying data element `spras`. These annotations are overlaid by the local annotations of simple type `ZBT_LanguageD`. You'll learn more about the resulting active annotations in Chapter 4, Section 4.3.1.

Simple types in CDS models

You can leverage simple types for typing fields and parameters (Section 2.16) of your CDS models. In Listing 2.9, CDS view field `Language1` is typed by simple type `ZBT_LanguageA`, and field `Language2` is typed by simple type `ZBT_LanguageD`. These fields take over the technical properties, including the annotations of their typing simple types.

```
define view entity Z_ViewWithSimpleTypes
  as select distinct from t000
{
  cast ( abap.lang'E' as ZBT_LanguageA preserving type ) as Language1,
  cast ( abap.lang'E' as ZBT_LanguageD preserving type ) as Language2
}
```

Listing 2.9 CDS View with Fields Typed by Simple Types

Incorporate Annotations by Simple Types into CDS Models

You can use simple types for incorporating field-specific or type-specific CDS annotations into your CDS models. In this context, simple types act as reusable metadata containers, which support you in efficiently defining consistent CDS models.

2.7 Case Statements

Case statements

You can use *case statements* for defining conditional calculations in your CDS view logic. Within a single case statement, you can define multiple execution paths via `when-then` switches. Based on the `when` condition, you can apply dedicated `then` instructions, which finally determine the calculated value of a field. At the end of a case statement, you can add an `else` instruction without conditions. This branch allows you to handle cases that aren't covered by the `when` conditions. Without this fallback valuation, the calculated field may have the value `null`, if none of the explicit `when` conditions match.

Avoid Non-Null Preserving Expressions

Unconditional `else` branches with assignments of constant values can result in non-null preserving expressions. Such expressions may prevent the SAP HANA optimizer from reordering the execution plan for the most efficient processing of the `select` statement, which in turn can result in performance issues.

Example: CDS view with case statements

Listing 2.10 provides an example for a CDS view with three case statements.

```
define view entity Z_ViewWithCaseStatements
  as select from ZI_SalesOrder
{
  key SalesOrder,
      case (SalesOrderType)
        when 'TAF' then 'X'
        when 'OAF' then 'X'
        else ''
      end                    as IsStandardOrder,
      cast( case (SalesOrderType)
        when 'TAF' then 'X'
        when 'OAF' then 'X'
        else ''
      end as abap.char(3) ) as IsStandardOrderAsChar3,
      case when SalesOrderType = 'TAF' then 'X'
           when SalesOrderType = 'OAF' then 'X'
        else ''
      end as IsStandardOrder2
}
```

Listing 2.10 CDS View with Case Statements

Example analysis

The first case statement checks the value of field `SalesOrderType` and sets the value of field `IsStandardOrder` to the constant `X` if the sales order type has a value of `TAF` or `OAF`. In all other cases, the value is set to the initial value. In the given case, the calculation implicitly results in a field of type `char` with length `1`.

You can influence the resulting type explicitly using a wrapping cast operation. This is illustrated for the second case statement, which uses the same logic as the first cast statement. Due to the enclosing cast operation, field `IsStandardOrderAsChar3` receives type `char` with length `3`.

The third case statement contains the same logic as the first case statement. From a principle perspective, the `when` conditions of the branches of this third case statement may also implement complex rules, for example, by applying pattern comparisons with `like` as well as by combining multiple conditions with `and` and `or` operators. In contrast, the `when` conditions in the first case statement only support simple comparisons of operands.

2.8 Session Variables

CDS session variables allow you to access information regarding the current runtime session within the CDS view logic. Similar to literal values, you can use session variables at various places within the implementation of your data selections.

Some standard usages are depicted in Listing 2.11, in which a filter is defined by and fields are derived from session variables.

Example: CDS view with session variables

```
define view entity Z_ViewWithSessionVariables
  as select from t000
{
  $session.client          as ClientField,
  $session.system_date     as SystemDateField,
  $session.system_language as SystemLanguageField,
```

```
  $session.user            as UserField,
  $session.user_date       as UserDateField,
  $session.user_timezone   as UserTimezoneField
}
where
  mandt = $session.client
```

Listing 2.11 CDS View with Session Variables

Table 2.2 provides you with an overview of the session variables that are supported by the CDS syntax.

Session Variable	Description	Correspondence in ABAP Code
`$session.client`	Current client	`sy-mandt`
`$session.system_date`	System date of the application server	`sy-datum`
`$session.system_language`	Logon language	`sy-langu`
`$session.user`	Current user	`sy-uname`
`$session.user_date`	Current user date	`sy-datlo`
`$session.user_timezone`	User time zone	`sy-zonlo`

Table 2.2 Session Variables

Session Variables versus Parameters

If you use session variables, the data selection of your CDS view is influenced by factors that a consumer of your CDS view may not be able to control explicitly. This can result in unwanted restrictions and make the selection results more difficult to interpret.

In many cases, you can replace session variables with parameters (Section 2.16), which eases the error analysis and maintenance of CDS views. However, introducing parameters may require the consumers of a CDS view to adapt. In contrast, the introduction of session variables is a local implementation detail of a view. Typically, from a technical perspective, it's compatible for ABAP consumers. Only when introducing a client dependency by using session variable `$session.client` may adaptions be required.

2.9 Client Handling

Standard data selection in ABAP

When performing standard selections on database tables via the ABAP SQL interface, client-specific data is automatically filtered by the current client of the ABAP session. To perform cross-client data accesses, either addition `using client` or addition `client specified` must be added to the `select` statement. Listing 2.12 shows an example of such a selection, where the data is read from client `001`.

```
SELECT *
   FROM zsalesorder
   CLIENT SPECIFIED
   INTO TABLE @DATA(lt_salesorder)
   WHERE client = '001'.
```

Listing 2.12 Cross-Client Data Selection in ABAP

Client handling in view entities

In CDS view entities, the standard client handling is enforced by the ABAP Data Dictionary infrastructure. This is achieved by automatically augmenting the technical view definition on SAP HANA with an additional `where` condition, if this is required.

Consider Restrictions Imposed by Access Controls

Cross-client data accesses via CDS views aren't allowed if the CDS views are protected by access controls (see Chapter 5).

Example: Client-dependent CDS view entity

Listing 2.13 and Listing 2.14 illustrate an example. The CDS view entity `ZI_Product` in Listing 2.13 selects from a client-dependent table `ZPRODUCT`.

```
define root view entity ZI_Product
  as select from zproduct
{
  key product            as Product,
      product_type       as ProductType,
      creation_date_time as CreationDateTime
}
```

Listing 2.13 CDS View Selecting from a Client-Specific Data Source

Even though it doesn't explicitly handle the client dependency, its generated database view maps the client field `CLIENT` onto context variable `CDS_CLIENT`, as depicted in Listing 2.14. This context variable is associated with the session variable `$session.client`, thus the selection result is restricted accordingly.

```
CREATE OR REPLACE VIEW "ZI_PRODUCT" AS SELECT
  "ZPRODUCT"."CLIENT" AS "MANDT",
  "ZPRODUCT"."PRODUCT" AS "PRODUCT",
  "ZPRODUCT"."PRODUCT_TYPE" AS "PRODUCTTYPE",
  "ZPRODUCT"."CREATION_DATE_TIME" AS "CREATIONDATETIME"
FROM "ZPRODUCT" "ZPRODUCT"
WHERE "ZPRODUCT"."CLIENT" = SESSION_CONTEXT(
  'CDS_CLIENT'
)
```

Listing 2.14 Create Statement of the Database View Generated from Listing 2.13

Client field

Because the client is fixed, the client field shouldn't be included in the projection lists of the CDS views. However, CDS views, whose records only differ in their client fields, are an exception.

The client field should also be contained in the signatures of CDS table functions (see Chapter 7) that access client-specific data records. For more information about this topic, see the ABAP documentation at *http://s-prs.co/v529403*.

2.10 Select Distinct Statements

Condense selection result

By applying the *select distinct statement* (`select distinct`), you can remove duplicate records from the selection result list and thus condense it. The comparison considers the values of all requested fields. Let's take a look at the sample CDS views in Listing 2.15 and Listing 2.16.

Example: CDS view without select distinct

CDS view `Z_ViewWithoutSelectDistinct` in Listing 2.15 uses table `T000`, which contains one record per client. Its implemented logic results in multiple selected records (one per client), which share the value `A` for field `Field1`. Correspondingly, no key can be specified for this CDS view.

```
define view entity Z_ViewWithoutSelectDistinct
 as select from t000
{
  abap.char'A' as Field1
}
```

Listing 2.15 CDS View without Select Distinct Statement

In contrast, CDS view `Z_ViewWithSelectDistinct` in Listing 2.16 combines the previous selection logic with a distinct statement. Independent from the number of setup clients, that is, the number of records provided by

table `T000`, CDS view `Z_ViewWithSelectDistinct` returns a single record (there is at least one record in table `T000`). Therefore, `Field1` can be marked as a key field.

```
define view entity Z_ViewWithSelectDistinct
 as select distinct from t000
{
  key abap.char'A' as Field1
}
```

Listing 2.16 CDS View with Select Distinct Statement

2.11 Union Views

Union views combine and unify data records of different data sources. The outcome of such a union is a results list that comprises all data records of the unified data sources and that is harmonized from the perspective of its consumers, providing them with uniform fields and associations. In Section 2.11.1, you get an overview about the fundamental modeling of union CDS views. In Section 2.11.2, we explain the differences between the plain `union` and the `union all` operators.

2.11.1 Union Definitions

Modeling union views

You define union views by combining multiple `select` statements with the `union` statement. Each individual selection branch must define the same fields and associations in the same order. In addition, the corresponding items of the selection lists must have the same definitions. This implies that the names of the elements within the individual branches must be the same. Furthermore, the underlying association definitions must specify the same `on` conditions, cardinalities, and target entities. The key definitions must be the same across all branches too. However, the types of the individual fields of the union view may deviate for each branch, if they are convertible. In such a case, the effective type of the field of the union view is derived from the corresponding field of the first `select` statement.

Similarly, element annotations are determined by the first selection branch. Therefore, only elements of the first selection branch may be annotated. Element annotations must not be propagated to a union view. This requires annotating the union view with `@Metadata.ignorePropagatedAnnotations: true`. Chapter 4, Section 4.3 explains the propagation logic in more detail.

Example: CDS views serving as data sources

Let's look at the following example, which comprises four CDS view definitions. CDS view Z_ViewAsDataSource**A** from Listing 2.17 acts as a data source for union CDS view Z_UnionView from Listing 2.20, later in this chapter.

```
define view entity Z_ViewAsDataSourceA
  as select distinct from t000
  association [0..1] to Z_ViewAsDataSourceC as _ViewC
    on $projection.FieldA3 = _ViewC.FieldC1
{
  key cast( 'A' as abap.char(1) ) as FieldA1,
      cast( 'B' as abap.char(1) ) as FieldA2,
      cast( 'C' as abap.char(2) ) as FieldA3,
      _ViewC
}
```

Listing 2.17 CDS View Z_ViewAsDataSourceA

Likewise, CDS view Z_ViewAsDataSource**B** from Listing 2.18 acts as a data source for union CDS view Z_UnionView from Listing 2.20.

```
define view entity Z_ViewAsDataSourceB
  as select distinct from t000
{
  key cast( 'B_X' as abap.char(3) ) as FieldB1,
      cast( 'A' as abap.char(1) )   as FieldB2
}
```

Listing 2.18 CDS View Z_ViewAsDataSourceB

Example: CDS view serving as association target

CDS view Z_ViewAsDataSource**C** from Listing 2.19 is used as the association target of view Z_ViewAsDataSource**A** from Listing 2.17 and union CDS view Z_UnionView from Listing 2.20.

```
define view entity Z_ViewAsDataSourceC
  as select distinct from t000
{
  key cast( 'C' as abap.char(2) )  as FieldC1,
      cast( 'C2' as abap.char(2) ) as FieldC2
}
```

Listing 2.19 CDS View Z_ViewAsDataSourceC

Analysis of the sample views

Each of the mentioned CDS views defines a single data record. For example, CDS view Z_ViewAsDataSource**A** selects a single data record from table T000, which is always populated, by applying the distinct statement. It returns

the constant values A, B, and C for its FieldA1, FieldA2, and FieldA3, respectively. Similarly, the other two CDS views, Z_ViewAsDataSource**B** and Z_ViewAsDataSource**C**, return single records with constant field values. Table 2.3 shows an overview of the data provided by these CDS views.

CDS View	FieldA1/B1/C1	FieldA2/B2/C2	FieldA3
Z_ViewAsDataSource**A**	A	B	C
Z_ViewAsDataSource**B**	B_X	A	–
Z_ViewAsDataSource**C**	C	C2	–

Table 2.3 Data Records of the CDS Views from Listing 2.17 to Listing 2.19

Example: Union CDS view

Union CDS view Z_UnionView in Listing 2.20 combines CDS views Z_ViewAsDataSource**A** and Z_ViewAsDataSource**B** as its data sources. In addition, it exposes association _ViewC to CDS view Z_ViewAsDataSource**C**.

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_UnionView
  as select from Z_ViewAsDataSourceA
  association [0..1] to Z_ViewAsDataSourceC as _ViewC on
$projection.UnionField1 = _ViewC.FieldC1
{
      @EndUserText.label: 'Label of UnionField1'
  key FieldA1 as UnionField1,
  key FieldA2 as UnionField2,
  key FieldA3 as UnionField3,
      _ViewC
}
union select from Z_ViewAsDataSourceB
  association [0..1] to Z_ViewAsDataSourceC as _ViewC on
$projection.UnionField1 = _ViewC.FieldC1
{
  key FieldB2  as UnionField1,
  key FieldB1  as UnionField2,
  key ''       as UnionField3,
      _ViewC
}
```

Listing 2.20 Union CDS View Z_UnionView

Analysis of sample union CDS view

To harmonize the element names, the fields contained in the individual selection branches of union CDS view Z_UnionView are mapped to field names UnionField1, UnionField2, and UnionField3 using alias function as. All

fields contained in the field list of the union CDS view have type `char` with length `1` or `2`. This definition is derived from the definition of underlying base fields `FieldA1`, `FieldA2`, and `FieldA3` of the first selection statement of the union CDS view. Table 2.4 shows the resulting records of the union CDS view.

UnionField1	UnionField2	UnionField3
A	B	C
A	B	Initial value

Table 2.4 Records of the Union CDS View Z_UnionView

Note that `FieldB1` of CDS view `Z_ViewAsDataSource`**B**, which has a maximum length of `3`, is implicitly shortened when it's mapped onto `UnionField1`. Therefore, the corresponding data record of the union CDS view contains value `B` instead of original value `B_X` (compare Table 2.3 with Table 2.4). If the types of the mapped fields aren't automatically mutually convertible, you must explicitly define a type conversion. This can typically be achieved by applying a `cast` operation (Section 2.4) for the fields in question.

In general, each selection branch of a union view must define the same number of fields, so `UnionField3` must also be inserted into the second selection branch. Because there is no related information available in underlying base CDS view `Z_ViewAsDataSource`**B**, this field is filled with the initial value, which corresponds to its type.

In the first selection branch, `UnionField1` is annotated with `@EndUserText.label`... Such annotations are valid for all other branches too; that is, the annotations of the first branch determine the corresponding annotations of the union CDS view as a whole.

Associations in union CDS views

In union CDS view `Z_UnionView` from Listing 2.20, association `_ViewC` to view `Z_ViewAsDataSource`**C** defined in its base view `Z_ViewAsDataSource`**A** can't simply be projected into the first selection branch because the definition of association `_ViewC` in the second selection branch of union view `Z_UnionView` has a different `ON` condition than in CDS view `Z_ViewAsDataSource`**A**. Due to the requirement that all selection branches must share the same association definitions in union views, this association must be redefined in the first selection branch too.

Two-layer union CDS views

To avoid redundancies in association definitions, you could split the definition of your union CDS views into two CDS views that are built one on top of the other. Listing 2.21 and Listing 2.22 show an example of such a two-layer construction.

Example: Union CDS view without association

CDS view `Z_UnionView`**WithoutAssociations** from Listing 2.21 implements `union` logic.

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_UnionViewWithoutAssociations
  as select from Z_ViewAsDataSourceA
{
      @EndUserText.label: 'Label of UnionField1'
  key FieldA1 as UnionField1,
  key FieldA2 as UnionField2,
  key FieldA3 as UnionField3
}

union select from Z_ViewAsDataSourceB
{
  key FieldB1 as UnionField1,
  key FieldB2 as UnionField2,
  key ''      as UnionField3
}
```

Listing 2.21 Union CDS View without Associations

Example: Dependent CDS view with association

CDS view `Z_UnionView`**WithAssociations** from Listing 2.22, which is based on this intermediate view, enriches `union` logic with association `_ViewC` without introducing redundancies in the definition.

```
define view entity Z_UnionViewWithAssociations
  as select from Z_UnionViewWithoutAssociations
  association [0..1] to Z_ViewAsDataSourceC as _ViewC
    on $projection.UnionField1 = _ViewC.FieldC1
{
  key UnionField1,
  key UnionField2,
  key UnionField3,
      _ViewC
}
```

Listing 2.22 CDS View with Association Based on Union CDS View

Association Definitions in Union CDS Views

If you need to define a larger number of associations within your union CDS view, it may be beneficial to relocate the association definitions to a separate superordinated CDS view. This CDS view can transfer the field list of the union CDS view via the regular projection mechanism and define the required associations once locally.

2.11.2 Union and Union All Logic

Union logic

When applying `union` logic, duplicate records, which originate from the different merged data sources, are automatically removed from the results list. For example, by applying `union` logic on top of the same data source `Z_ViewAsDataSource`**`A`** from Listing 2.17, the implementation of CDS view `Z_UnionView`**`WithoutDuplicates`** from Listing 2.23 leads to a single resulting data record.

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_UnionViewWithoutDuplicates
  as select from Z_ViewAsDataSourceA
{
  key FieldA1
}
union select from Z_ViewAsDataSourceA
{
  key FieldA1
}
```

Listing 2.23 Union CDS View without Duplicate Records (Union Logic)

Union All logic

If this isn't desired, you should use `union all` logic, which retains the data records of the data sources involved. In contrast to the selection in Listing 2.23, by applying `union all` logic, the selection result of CDS view `Z_UnionView`**`WithDuplicates`** from Listing 2.24 contains two identical data records.

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_UnionViewWithDuplicates
  as select from Z_ViewAsDataSourceA
{
  FieldA1
}
union all select from Z_ViewAsDataSourceA
{
  FieldA1
}
```

Listing 2.24 Union CDS View with Duplicate Records (Union All Logic)

Performance Implications

Due to the missing comparison and filtering of data records, `union all` logic typically outperforms `union` logic. If applicable, you should therefore choose the `union all` logic.

Key definitions

Merging selection results from various data sources requires a critical check of the key definitions of the union CDS views. If CDS views that have their own key definitions are merged, the key of the union CDS view isn't necessarily the same as the superset of all key fields of the merged CDS views. In union CDS view `Z_UnionView` from Listing 2.20, not only are `UnionField1` and `UnionField2` part of the key but also `UnionField3`. In the given example, `UnionField3` is required for a unique differentiation between the two data records of the selection result (refer to Table 2.4). If the second selection branch of the union view would set this field to value `C` instead of setting it to the initial value, both selection branches would return the same data record. In this case, you wouldn't be able to designate a unique key for union CDS view `Z_UnionView` as a whole.

In general, a unique key can't be specified for a CDS view with duplicate records. Therefore, the key definition should always be omitted in such a CDS view (refer to Listing 2.24).

2.12 Intersect and Except Statements

Besides merging records of data sources, the CDS syntax also supports you in defining intersections of and exceptions from data sources, which will be illustrated in this section.

Example: CDS views serving as data sources

Listing 2.25 shows CDS view `Z_UnionViewAsDataSource`**`A`**, which will act as a data source. It returns two records as given in Table 2.5.

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_UnionViewAsDataSourceA
  as select distinct from t000
{
  key 'A' as Field1
}
union all select distinct from t100 {
  key 'B' as Field1
}
```

Listing 2.25 CDS View Z_UnionViewAsDataSourceA

Z_UnionView-AsDataSourceA	Z_UnionView-AsDataSourceB	Z_ViewWith-Intersect	Z_ViewWith-Except
A	A	A	B
B	C	-	-

Table 2.5 Values of Field1 of the CDS Views Records from Listing 2.25 to Listing 2.28

Similarly, Listing 2.26 shows CDS view Z_UnionViewAsDataSource**B**, which will act as another data source. It also returns two records, as given in Table 2.5.

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_UnionViewAsDataSourceB
  as select distinct from t000
{
  key 'A' as Field1
}
union all select distinct from t100 {
  key 'C' as Field1
}
```

Listing 2.26 CDS View Z_UnionViewAsDataSourceB

Example: Intersection

If you want to determine those records that two data sources have in common, you can use the CDS syntax element `intersect`. CDS view `Z_ViewWithIntersect` from Listing 2.27 defines such an intersection of CDS view Z_UnionViewAsDataSource**A** from Listing 2.26 and CDS view Z_UnionViewAsDataSource**B** from Listing 2.25. Both these data sources contain a record with the value `A` for `Field1`, which represents the result of the intersection (refer to Table 2.5).

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_ViewWithIntersect
  as select from Z_UnionViewAsDataSourceA
{
  key Field1
}
intersect select from Z_UnionViewAsDataSourceB
{
  key Field1
}
```

Listing 2.27 CDS View with Intersect

Example: Exception

If you want to determine those records that are only contained in one of two data sources, you can use the CDS syntax element `except`. CDS view `Z_ViewWithExcept` from Listing 2.28 excepts records of CDS view Z_UnionViewAsDataSource**B** from Listing 2.25 from CDS view Z_UnionViewAsDataSource**A** from Listing 2.24. Because both these data sources contain a record with the value `A` for `Field1`, the result is defined by the record of CDS view Z_UnionViewAsDataSource**A** with the value `B` for `Field1` (refer to Table 2.5).


```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_ViewWithExcept
  as select from Z_UnionViewAsDataSourceA
{
  key Field1
}
except select from Z_UnionViewAsDataSourceB
{
  key Field1
}
```

Listing 2.28 CDS View with Except

2.13 Joins

Joins allow you to model conditional links between two data sources. The join conditions describe criteria for linking a data record of the primary data source with a data record of the secondary data source.

Using joins

You can use elements of joined data sources for defining elements in the projection list of your CDS views. In addition, you can use the elements of joined data sources for enriching the `where` conditions of your CDS views.

> **Performance Aspects**
>
> Joins make optimizing the execution plan of the selection request at the database level more difficult. Therefore, you should only integrate those joins into the logic of your CDS view definition that are essentially required for the view's functionality, and avoid unnecessary joins of data sources resulting in denormalized CDS view models. This particularly applies to CDS views that you define for reuse purposes. Instead, you should consider providing appropriate associations between the CDS models. These associations can be used by the consumers of your model to enrich the data records in a convenient way, where this is actually necessary. We'll discuss associations further in Chapter 3.

Types of joins

CDS views support four flavors of joins:

- **Left outer joins**
  These joins relate records of a primary data source with records of a secondary data source so that the result contains all the data records of the primary data source.
- **Right outer joins**
  These joins relate records of the secondary data source with the records

of the primary data source so that the result contains all the data records of the secondary data source.

- **Inner joins**
  These joins relate records of a primary data source with records of a secondary data source so that the result contains only those records of the primary data source for which at least one join partner in the secondary data source exists.
- **Cross joins**
  These joins combine all records of a primary data source with all records of a secondary data source. The number of records in the result set is equal to the number of records of the primary data source multiplied by the number of records of the secondary data source.

Cardinality of joins

When dealing with joins, you'll often be confronted with the term *cardinality*. The cardinality of the join partner specifies the number of data records that result from the join relationship. For example, if there is more than one corresponding data record in the secondary data source for a single record of the primary data source, the number of resulting data records is multiplied by the cardinality of the join partner when applying the left outer join logic.

Left outer and inner joins

In the following discussion, we'll focus on the left outer joins and inner joins, which are the most common join types. Sample CDS views `Z_ViewWithLeftOuterJoins` from Listing 2.31 and `Z_ViewWithInnerJoins` from Listing 2.33 (later in this section) will illustrate the various aspects of the join logic.

Example: CDS views serving as data sources

Both these CDS views use CDS view `Z_ViewAsDataSourceD` from Listing 2.29 as their primary data source.

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_ViewAsDataSourceD
  as select distinct from t000
{
  key cast( 'A' as abap.char(1) ) as FieldD1,
      cast( 'D' as abap.char(1) ) as FieldD2
}
union select distinct from t000
{
  key cast( 'C' as abap.char(1) ) as FieldD1,
      cast( 'E' as abap.char(1) ) as FieldD2
}
```

Listing 2.29 CDS View Z_ViewAsDataSourceD

CDS view `Z_ViewAsDataSourceE` from Listing 2.30 acts as their secondary data source.

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_ViewAsDataSourceE
  as select distinct from t000
{
  key cast( 'D' as abap.char(1) ) as FieldE1,
  key cast( 'H' as abap.char(1) ) as FieldE2
}
union select distinct from t000
{
  key cast( 'D' as abap.char(1) ) as FieldE1,
  key cast( 'I' as abap.char(1) ) as FieldE2
}
union select distinct from t000
{
  key cast( 'F' as abap.char(1) ) as FieldE1,
  key cast( 'I' as abap.char(1) ) as FieldE2
}
```

Listing 2.30 CDS View Z_ViewAsDataSourceE

CDS views `Z_ViewAsDataSourceD` and `Z_ViewAsDataSourceE` return two or three data records, as shown in Table 2.6.

CDS View	FieldD1	FieldD2	FieldE1	FieldE2
`Z_ViewAsDataSourceD`	A	D	–	–
	C	E	–	–
`Z_ViewAsDataSourceE`	–	–	D	H
	–	–	D	I
	–	–	F	I
`Z_ViewWithLeftOuterJoins`	A	D	–	H
	A	D	–	I
	C	E	–	null
`Z_ViewWithInnerJoins`	A	D	–	H
	A	D	–	I

Table 2.6 Records of the Data Sources and the Join CDS Views Built on Top

Example: CDS view with left outer join

If these two CDS views are linked as data sources by a left outer join according to CDS view `Z_ViewWithLeftOuterJoins` from Listing 2.31, the results list comprises three data records (see Table 2.6).

```
define view entity Z_ViewWithLeftOuterJoins
  as select from           Z_ViewAsDataSourceD
    left outer one to many join Z_ViewAsDataSourceE
      on Z_ViewAsDataSourceD.FieldD2 = Z_ViewAsDataSourceE.FieldE1
{
  key Z_ViewAsDataSourceD.FieldD1,
  key Z_ViewAsDataSourceD.FieldD2,
  key Z_ViewAsDataSourceE.FieldE2
}
```

Listing 2.31 CDS View with a Left Outer Join

Example analysis

The first entry of CDS view `Z_ViewAsDataSourceD`, which has value `A` for key field `FieldD1`, is linked to two data records of CDS view `Z_ViewAsDataSourceE`. Consequently, the results list of CDS view `Z_ViewWithLeftOuterJoins` has two data records for this entry. This potential increase of the cardinality is expressed by addition `to many` in the left outer join statement. If you take a closer look at the values of `FieldE1` of the join partner CDS view `Z_ViewAsDataSourceE`, you'll recognize that there is no or a single (i.e., at maximum one) record of CDS view `Z_ViewAsDataSourceD` with a matching value of `FieldD1`. Accordingly, source cardinality of the left outer join in CDS view `Z_ViewWithLeftOuterJoins` is defined as `one`. Overall, this results in a one-to-many join.

[+]

Specify the Cardinality of Joins

You should always specify the maximum target cardinality (`to one` or `to exact one` or `to many`) of a join partner. Furthermore, it may be useful to define the source cardinality (`one` or `exact one` or `many`) too. This specification is used both to document the composition of the CDS view and optimize the processing of a selection request in the database. However, if you maintain the cardinality information, you should make sure that it's defined correctly; otherwise, the selection result may become incorrect!

The second record of view `Z_ViewAsDataSourceD`, which has value `C` for key field `FieldD1`, doesn't have a join partner. According to the semantics of the left outer join, this data record remains in the results list. However, in this case, associated `FieldE2` of CDS view `Z_ViewWithLeftOuterJoins` receives value `null` on the database level.

Null and initial values

If you transfer the selection result to an internal ABAP table according to Listing 2.32, the `null` value is implicitly converted to the initial value of the corresponding ABAP field. By default, the ABAP runtime environment doesn't distinguish between initial values and `null` values in contrast to the CDS language and the SQL logic.

```
SELECT *
   FROM z_viewwithleftouterjoins
   INTO TABLE @DATA(lt_viewwithleftouterjoins).
```

Listing 2.32 Data Selection from the CDS View from Listing 2.31 in ABAP

[!]

Required Handling of Null Values

On the SQL level, the difference between initial values and `null` values is significant in that `null` values represent nonvaluated data. In the CDS views, `null` values usually result from left outer joins without a matching join partner.

When modeling your CDS view logic, you must consider this distinction and handle it if necessary. For example, when comparing field values in `where` conditions, you have to consider potential `null` values. However, `null` values won't simply be replaced by initial values in your CDS logic. Otherwise, you may experience significant performance issues.

Inner joins

In contrast to a left outer join, an inner join removes source records without a join partner from the selection result.

Example: CDS view with inner join

Accordingly, the selection result of CDS view `Z_ViewWithInnerJoins` from Listing 2.33 contains only two data records (refer to Table 2.6).

```
define view entity Z_ViewWithInnerJoins
  as select from Z_ViewAsDataSourceD
    inner join   Z_ViewAsDataSourceE
      on Z_ViewAsDataSourceD.FieldD2 = Z_ViewAsDataSourceE.FieldE1
{
  key Z_ViewAsDataSourceD.FieldD1,
  key Z_ViewAsDataSourceD.FieldD2,
  key Z_ViewAsDataSourceE.FieldE2
}
```

Listing 2.33 CDS View with an Inner Join

Left outer joins without null values

You can achieve a corresponding selection result using a left outer join relationship too. In this case, the corresponding left outer join statement must be accompanied by additional `where` conditions, which ensure that a data

record of the respective join partner exists. Within these `where` conditions, you can validate that an adequate field (usually a key field) of the join partner has a value, which is different from the `null` value. Listing 2.34 illustrates this approach by checking `FieldE2` of left outer join partner CDS view `Z_ViewAsDataSourceE`.

```
define view entity Z_ViewWithLeftOuterJoinsFiltrd
  as select from            Z_ViewAsDataSourceD
    left outer one to many join Z_ViewAsDataSourceE
      on Z_ViewAsDataSourceD.FieldD2 = Z_ViewAsDataSourceE.FieldE1
{
  key Z_ViewAsDataSourceD.FieldD1,
  key Z_ViewAsDataSourceD.FieldD2,
  key Z_ViewAsDataSourceE.FieldE2
}
where
  Z_ViewAsDataSourceE.FieldE2 is not null
```

Listing 2.34 CDS View with a Left Outer Join and an Additional Existence Check for the Join Partner

Qualifying elements with their data sources

If you combine multiple data sources using joins, you must ensure that references to the elements of these data sources remain unique. To achieve this, you have to prefix the element name with the name of its data source, separated by a period. In the previous examples, this was realized by using the name of the data source as a qualifier. However, if the same data source is included multiple times in the same CDS view, you have to specify suitable alias names for each of its occurrences, as shown in Listing 2.35. In this case, the CDS view `Z_ViewAsDataSourceD` is used twice as a data source. To distinguish between the two embedments of this CDS view, they are assigned distinct alias names `D1` and `D2`.

```
define view entity Z_ViewWithJoinsAndDataSrcAlias
  as select from           Z_ViewAsDataSourceD as D1
    left outer exact one to exact one join Z_ViewAsDataSourceD as D2
      on D1.FieldD1 = D2.FieldD1
{
  key D1.FieldD1,
      D2.FieldD2
}
```

Listing 2.35 CDS View with Alias Names for the Same Data Source Joined Multiple Times

2.14 SQL Aggregation Functions

SQL aggregation functions allow you to perform calculations of predefined aggregates efficiently on the database level. You can use these SQL functions within the implementation of your CDS views.

Define aggregations

To do this, first define the aggregation level to which you want to aggregate the result. Enter the aggregation level using syntax element `group by`, which should be placed after the projection list. It has to include all fields of the data sources that are included in the projection list of the CDS view. Fields whose aggregate is calculated are excluded from this. Within the projection list of the CDS view, you then apply the envisioned aggregation function to the fields that are relevant for aggregation.

Example: Aggregation

An example is given by CDS view `Z_ViewAsDataSourceF` from Listing 2.36 and CDS view `Z_ViewWithAggregations` from Listing 2.37.

Example: CDS view serving as a data source

CDS view `Z_ViewAsDataSourceF` acts as the data source for the aggregation.

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_ViewAsDataSourceF
  as select distinct from t000
{
  key abap.char'A' as Field1,
  key abap.char'A' as Field2,
      abap.int1'1' as Field3
}
union all select distinct from t000
{
  key abap.char'A' as Field1,
  key abap.char'B' as Field2,
      abap.int1'2' as Field3
}
union all select distinct from t000
{
  key abap.char'A' as Field1,
  key abap.char'C' as Field2,
      abap.int1'3' as Field3
}
```

Listing 2.36 CDS View Z_ViewAsDataSourceF

It defines three data records, as depicted in Table 2.7.

Field1	Field2	Field3
A	A	1
A	B	2
A	C	3

Table 2.7 Records of CDS View Z_ViewAsDataSourceF

Example: Aggregating CDS view

CDS view `Z_ViewWithAggregations` from Listing 2.37 contains the aggregation logic. Its `group by` statement defines that the data source records are to be aggregated up to the level of the first key field: `Field1`.

```
define view entity Z_ViewWithAggregations
  as select from Z_ViewAsDataSourceF
{
  key Field1,
      min(Field3)                     as FieldWithMin,
      max(Field3)                     as FieldWithMax,
      avg( Field3 as abap.decfloat34 ) as FieldWithAvg,
      cast( sum(Field3) as abap.int4 ) as FieldWithSum,
      count( distinct Field1 )        as FieldWithCountDistinct,
      count(*)                        as FieldWithCountAll
}
  group by Field1
```

Listing 2.37 CDS View with Aggregation Functions

Example analysis

The result of the selection from CDS view `Z_ViewWithAggregations` comprises a single data record. The minimum (`min`) value (`FieldWithMin = 1`), maximum (`max`) value (`FieldWithMax = 3`), average (`avg`) value (`FieldWithAvg = 2`), and summed-up (`sum`) value (`FieldWithSum = 6`) of `Field3` are calculated. In addition, for each entry in the results list of aggregating view `Z_ViewWithAggregations`, the number (= 1) of its underlying aggregated original data records that differs in the value of grouping field `Field1` (`count distinct`) is calculated and returned by `FieldWithCountDistinct`. Finally, the total number (= 3) of underlying aggregated data records (`count(*)`) is calculated and returned by `FieldWithCountAll`.

Aggregated field types

Note that the fields for minimum value `FieldWithMin` and maximum value `FieldWithMax` retain type `int1` of underlying base field `Field3`. By default, this would also hold true for field `FieldWithSum`, which contains the result of the summation. However, in the example, a cast operation is applied, which explicitly extends the type to `int4`. When applying the average function, an

explicit type assignment is required. `FieldWithAvg` is typed as `decfloat34`. The fields that expose the result of the count operation—`FieldWithCountDistinct` and `FieldWithCountAll`—are implicitly assigned ABAP type `int4`.

Avoid Memory Overflows

To avoid a memory overflow at the execution time of the data selection when using aggregation functions, you should check the expected value ranges and existing type assignments of the fields and enhance these types by explicit type conversions if necessary.

Performance aspects

If you use aggregation functions, the grouping fields defined in the `group by` statement ideally should not contain any calculated or joined fields. Furthermore, it may be beneficial to perform the required aggregations as deep as possible within the CDS view stack to reduce the amount of processed data to its minimum as soon as possible.

Using conversion functions

If the values to be aggregated represent amounts or quantities, you must normalize them before the aggregation is performed. This means that you may first have to convert them to the same unit or keep the currency and unit information in the aggregation result to avoid unwanted calculation errors. We'll explain conversion functions in Section 2.18.

Performance Aspects

If you use conversion functions on nonaggregated single data records, this can have a negative effect on performance. In such cases, you should consider first aggregating the data records while keeping their units and applying the conversion functions to the preaggregated data records.

2.15 Projection Fields

Projection fields are defined in the select list of a CDS view. Within the CDS view definition, you can access these fields by prefixing their names with operator `$projection` followed by a dot. You can use projection fields to refer to calculation results defined within the same view. This allows you to build up nested calculations with the exposed fields serving as reusable intermediate results. In addition, you can use projection fields (with some restrictions) for defining associations (see Chapter 3).

Example: CDS view with projection fields

Listing 2.38 illustrates the implementation of CDS view `Z_ViewWithProjectionFields`, which leverages projection fields for further calculations.

```
define view entity Z_ViewWithProjectionFields
  as select distinct from t000
{
  abap.char'A' as FieldA,
  abap.char'B' as FieldB,
  concat( $projection.FieldA, $projection.FieldB ) as FieldC,
  concat( abap.char'A', abap.char'B' ) as FieldC2,
  concat( $projection.FieldC, abap.char'D' ) as FieldD,
  concat( concat( abap.char'A', abap.char'B' ), abap.char'D' )
    as FieldD2
}
```

Listing 2.38 CDS View with Projection Fields

Example analysis

In this example, two fields, `FieldA` and `FieldB`, are defined with constant values `A` and `B`, respectively. These locally defined fields are concatenated to `FieldC`. The resulting value is the same as when concatenating the original constants, `A` and `B`; that is, `FieldC2` holds the same value as `FieldC`. Similarly, `FieldD` and `FieldD2` hold the same values, with `FieldD` reusing the determination logic of `FieldC` and with `FieldD2` calculating the result from scratch.

From a technical perspective, the modeled reuse on the CDS level is being expanded when creating the corresponding database view on SAP HANA. This is shown by the `create` statement in Listing 2.39, in which the definitions of fields `FieldC` and `FieldC2`—which are `FieldD` and `FieldD2`, respectively—are identical.

```
CREATE OR REPLACE VIEW "Z_VIEWWITHPROJECTIONFIELDS" AS SELECT
  DISTINCT N'A' AS "FIELDA",
  N'B' AS "FIELDB",
  CONCAT(
    N'A',
    N'B'
  ) AS "FIELDC",
  CONCAT(
    N'A',
    N'B'
  ) AS "FIELDC2",
  CONCAT(
    RTRIM(
      CONCAT(
        N'A',
        N'B'
      )
    ),
```



```
    N'D'
  ) AS "FIELDD",
  CONCAT(
    RTRIM(
      CONCAT(
        N'A',
        N'B'
      )
    ),
    N'D'
  ) AS "FIELDD2"
FROM "T000" "T000"
```

Listing 2.39 Create Statement of the Database View Generated from Listing 2.38

2.16 Parameters

Parameters are constituents of the CDS model signature. They represent scalar input values that must be supplied by the caller when performing data selections.

Using parameters

You can access and evaluate parameter values in the logic of your CDS models. This allows you to equip consumers of your CDS models with predefined control options for their data selections. You can also use parameters for enforcing a restriction of the selection result by applying them as predefined filter criteria.

Definition of parameters

Parameters are listed directly after the name of the CDS model. The parameter list is introduced by CDS syntax element `with parameters`. Each parameter has a name and a type separated by a colon. The type can be based on elementary ABAP types, data elements, and simple types.

The example from Listing 2.40 shows two alternatives for assigning types to parameters.

```
define view entity Z_ViewWithParameters
  with parameters
    P_KeyDate  : abap.dats,
    P_Language : sylangu
  as select from …
```

Listing 2.40 Definition of Parameters

The parameter names must be unique within a CDS model and must be different from the element names.

Parameter Names

You should start the name of parameters with the prefix `P_`, followed by the semantic name of the transported value. Following this rule creates a clear separation of the parameter names from the names of the remaining elements of the CDS models.

Parameters in CDS views

Parameters can be used in different places within a CDS model. For example, you can use parameters to do the following:

- Define `where` conditions in a CDS view.
- Calculate fields in the projection list of a CDS view.
- Supply parameters of other data sources with the requested values.

Within the implementation of a CDS model, parameters can be accessed by the preceding syntax element `$parameters`, which is separated by a dot from the parameter name.

Let's now have a look at the potential usages of parameters with the following sample CDS views and ABAP code snippets.

Example: CDS view serving as a data source

In all depicted cases, CDS view `Z_ViewWithParametersDataSource` from Listing 2.41 serves as the fundamental data source. According to its definition, it returns a single data record.

```
define view entity Z_ViewWithParametersDataSource
  as select distinct from t000
{
  key abap.char'A'        as KeyField,
  key abap.lang'E'        as Language,
  key abap.dats'99580809' as ValidityEndDate,
      abap.dats'20111004' as ValidityStartDate
}
```

Listing 2.41 CDS View Z_ViewWithParametersDataSource

Example: CDS view with parameters

CDS view `Z_ViewWithParameters` in Listing 2.42 uses CDS view `Z_ViewWithParametersDataSource` as its direct base data source. It defines parameters for key date `P_KeyDate` and language `P_Language`.

```
define view entity Z_ViewWithParameters
  with parameters
    P_KeyDate  : abap.dats,
    P_Language : sylangu
  as select from Z_ViewWithParametersDataSource
```



```
  association [0..*] to Z_ViewWithParametersAscTarget as _Target
    on  $projection.KeyField = _Target.KeyField
  association [0..1] to Z_ViewWithParametersAscTarget as _FilteredTarget
    on  $projection.KeyField = _FilteredTarget.KeyField
    and $projection.Language = _FilteredTarget.Language
{
  key KeyField,
      ValidityEndDate,
      ValidityStartDate,
      $parameters.P_Language     as Language,
      _Target(P_ValidityDate: $parameters.P_KeyDate)
        [1:Language= $parameters.P_Language].KeyField as TargetKeyField,
      _FilteredTarget
}
where ValidityEndDate   >= $parameters.P_KeyDate
  and ValidityStartDate <= $parameters.P_KeyDate
  and Language          =  $parameters.P_Language
```

Listing 2.42 CDS View with Parameters

Example analysis

These parameters are used in its `where` condition for restricting the selection result. Parameter `P_Language` is also added as field `Language` to the projection list of the CDS view. This field will have the same value as the parameter for all data records. As a result, the parameter value can indirectly be used in the definition of association `_FilteredTarget` in expression `$projection.Language = _FilteredTarget.Language`. Alternatively, you could use the parameter directly on the right-hand side of an expression such as `_FilteredTarget.Language = $parameters.P_Language`.

Example: Association target with parameter

The target of association `_FilteredTarget` is CDS view `Z_ViewWithParametersAscTarget` with parameter `P_ValidityDate`, as shown in Listing 2.43.

```
define view entity Z_ViewWithParametersAscTarget
  with parameters
    P_ValidityDate : abap.dats
  as select from Z_ViewWithParametersDataSource
{
  key KeyField,
  key Language,
      ValidityEndDate,
      ValidityStartDate
}
where ValidityEndDate   >= $parameters.P_ValidityDate
  and ValidityStartDate <= $parameters.P_ValidityDate
```

Listing 2.43 CDS View with a Parameter Serving as an Association Target

Parameters in path expressions

When accessing field `KeyField` of this target CDS view by applying path expression `_Target(P_ValidityDate: $parameters.P_KeyDate)[1:Language=$parameters.P_Language].KeyField` in CDS view `Z_ViewWithParameters`, its own parameter `P_ValidityDate` must be supplied. This is achieved by binding it to parameter `P_KeyDate` of CDS view `Z_ViewWithParameters`.

Within the depicted path expression, the second local parameter `P_Language` of the source view `Z_ViewWithParameters` is also used for applying a filter condition on associated target field `Language`. This means that the key and thus the associated data record of target CDS view `Z_ViewWithParametersAscTarget` are uniquely identified. As a result, the maximum cardinality of the path expression is defined as `1`.

Missing Support for Specifying Target Parameters in Association Definitions

Parameters of association targets can't be supplied within the definition of an association. Among other things, this means you can't explicitly bind parameters of associated CDS models. Therefore, a coupling of the parameters in CDS models can't be enforced. Instead, the consumer has to supply the parameters consistently when selecting data from the associated CDS models.

Example: Parameters in another CDS view

If you want to use CDS model `Z_ViewWithParameters` as a data source within a `select` statement, its parameters must be supplied with values. For example, Listing 2.44 shows CDS view `Z_ViewWithParametersConsumer`, which supplies parameters `P_KeyDate` and `P_Language` of its data source with session variable `$session.system_date` and constant value `E`, respectively.

```
define view entity Z_ViewWithParametersConsumer
  as select from Z_ViewWithParameters(
    P_KeyDate: $session.system_date,
    P_Language: 'E')
{
  key KeyField
}
```

Listing 2.44 CDS View Supplying Parameters of Its Data Source

Example: Parameters in ABAP

Listing 2.45 shows a corresponding selection in ABAP code. In this example, ABAP system field `sy-datum` is used for supplying parameter `P_KeyDate`.

```
SELECT keyfield
   FROM z_viewwithparameters(
     p_keydate = @sy-datum,
     p_language = 'E' )
   INTO TABLE @DATA(lt_viewwithparameters).
```

Listing 2.45 Data Selection in ABAP Corresponding to the CDS Modeling in Listing 2.44

Mandatory and optional supply of parameter values

Within the CDS view stack, parameters must always be supplied. In contrast, the ABAP SQL interface supports the automatic supply of parameters, which are annotated with `@Environment.systemField:...`. The permitted values of this annotation, such as `SYSTEM_DATE` and `SYSTEM_LANGUAGE`, correspond to the values of the ABAP system fields, such as `sy-datum` and `sy-langu`. If such an annotated parameter isn't supplied explicitly with a value when selecting data via the ABAP SQL interface, the ABAP runtime automatically fills the parameter with the value of the corresponding ABAP system field. As a result, the corresponding parameter of the CDS model becomes optional from an ABAP perspective.

Example: Data selection in ABAP

This is demonstrated by the following example, which comprises the CDS view `Z_ViewWithOptionalParameters` in Listing 2.46 and the two data selections in Listing 2.47 and Listing 2.48.

```
define view entity Z_ViewWithOptionalParameters
  with parameters
    @Environment.systemField: #SYSTEM_DATE
    P_KeyDate : abap.dats
  as select distinct from Z_ViewWithParametersDataSource
{
  key KeyField
}
where ValidityEndDate   >= $parameters.P_KeyDate
  and ValidityStartDate <= $parameters.P_KeyDate
```

Listing 2.46 CDS View with a Specially Annotated Parameter

Parameter `P_KeyDate` of CDS view `Z_ViewWithOptionalParameters` is annotated with `@Environment.systemField: #SYSTEM_DATE`. It's explicitly supplied in Listing 2.47 with system field `sy-datum`.

```
SELECT *
   FROM z_viewwithoptionalparameters( p_keydate = @sy-datum )
   INTO TABLE @DATA(lt_viewwithoptionalparameters).
```

Listing 2.47 Data Selection in ABAP with Explicit Supply of the CDS View Parameter

In contrast, the data selection in Listing 2.48 doesn't supply this parameter explicitly.

```
SELECT *
   FROM z_viewwithoptionalparameters
   INTO TABLE @DATA(lt_viewwithoptionalparameters).
```

Listing 2.48 Data Selection in ABAP with Implicit Supply of the CDS View Parameter

Nevertheless, both selections yield the same result.

Check Usage of Parameters

Because parameters typically force consumers of a CDS model to supply them with values, the consumers must know the lists of permitted values to carry out data selections successfully. Therefore, a simple data selection may not be possible without this knowledge. Furthermore, the addition, modification, or removal of a parameter is usually incompatible for consumers of the affected CDS model because consumers have to react to these changes. Therefore, you should only define parameters in your CDS models if they are essential for the functionality of the corresponding CDS models.

2.17 Reference Fields

Reference fields are amount and quantity fields that define references to currency and unit fields, respectively. Such a reference is technically enforced for fields that have the ABAP type `curr` or `quan`.

Defining amount and quantity fields

In CDS models, references to currencies and units are established by means of annotations `@Semantics.amount.currencyCode...` and `@Semantics.quantity.unitOfMeasure...`, respectively. Therein, the assigned annotation values represent the respective currency and unit fields that the annotated fields refer to. The referenced currency and unit fields with fields of type `curr` and `quan` have to be defined as fields of type `cuky` or `unit`. The currency codes themselves originate from table `TCURC`. The quantity units are defined by table `T006`.

Example: CDS view with amount and quantity fields

Listing 2.49 shows sample CDS view `Z_ViewWithReferenceFields`**A** with quantity and amount fields.

```
define view entity Z_ViewWithReferenceFieldsA
  as select distinct from t000
{
```



```
  @Semantics.quantity.unitOfMeasure: 'QuantityUnit'
  abap.quan'1234.56'                          as Quantity,
  abap.unit'PC'                               as QuantityUnit,
  @Semantics.amount.currencyCode: 'Currency'
  abap.curr'1234.56'                          as Amount,
  abap.cuky'USD'                              as Currency
}
```

Listing 2.49 CDS View with Amount and Quantity Fields

Use Curr Fields with Two Decimals Only

When defining `curr` fields, you should always define them with two decimals. Otherwise, the currency shift logic may not work properly.

You'll learn more about the currency shift logic in Chapter 17, Section 17.2.3.

Calculated quantities

Besides plain quantities referring to units of table `T006`, the CDS models also support defining quantities that are calculated from other quantities and amounts. For regular quantities, for example, the units of such calculated quantities are referenced by `@Semantics.quantity.unitOfMeasure...`. These units can represent complex expressions involving various base units and currencies. Accordingly, they are defined as character fields. Typically, such fields are modeled as virtual fields in the CDS models and populated by ABAP code. Of course, if technically feasible, you could also construct the unit values by applying suitable string operations, as depicted earlier in Listing 2.46.

Type conversions of reference fields

If a `curr` field will be incorporated into a calculation, it first must be converted into a plain value without its currency reference by using function `get_numeric_value`. Similarly, if you want to decouple quantities from their unit, you can use the same function. For a `curr` field, this function not only removes the reference and converts the `curr` value to a value of type `decfloat34` but also applies an implicit decimal shift.

If you only want to convert the type of an amount field from `curr` to `decfloat` while keeping its reference field nature, you can use function `curr_to_decfloat_amount`. Because the resulting field is to be considered an amount, it has to specify a reference to its currency via `@Semantics.amount.currencyCode...`.

Example: CDS view with type conversions of reference fields

CDS view `Z_ViewWithReferenceFields`**B** from Listing 2.50, which is based on CDS view `Z_ViewWithReferenceFields`**A** from Listing 2.49, illustrates the usage of the aforementioned conversion functions and the definition of a calculated quantity, including its unit.

```
define view entity Z_ViewWithReferenceFieldsB
  as select from Z_ViewWithReferenceFieldsA
{
  @Semantics.quantity.unitOfMeasure: 'QuantityUnit'
  Quantity,
  QuantityUnit,
  @Semantics.amount.currencyCode: 'Currency'
  Amount,
  Currency,
  get_numeric_value( Amount )       as AmountWithoutReference,
  get_numeric_value( Quantity )     as QuantityWithoutReference,
  @Semantics.amount.currencyCode: 'Currency'
  curr_to_decfloat_amount( Amount ) as DecfloatAmount,
  @Semantics.quantity.unitOfMeasure: 'CalculatedUnit'
  get_numeric_value( Amount ) / $projection.quantity
                                    as AmountPerQuantity,
  cast( concat( Currency,
                concat( '/', QuantityUnit)
    ) as abap.char(50) )   as CalculatedUnit
}
```

Listing 2.50 CDS View with Converted Amount and Quantity Fields

Display reference information

If you want to get an overview about the reference fields of your CDS entities, you can place the cursor on the CDS entity name and press F2. This will open up a popup that lists the fields along with their types and references, as illustrated in Figure 2.2 for CDS view `Z_ViewWithReferenceFieldsB` from Listing 2.50.

Z_ViewWithReferenceFieldsB (View)
View with Reference Fields

Client Handling
Client dependent
Client session variable used

Column	*Data Element*	*Data Type*	*Unit/Currency Reference*	*Description*
Quantity		quan(6,2)	quantityunit	
QuantityUnit		unit(2)		
Amount		curr(6,2)	currency	
Currency		cuky(5)		
AmountWithoutReference		decfloat34(34)		
QuantityWithoutReference		decfloat34(34)		
DecfloatAmount		decfloat34(34)	currency	
AmountPerQuantity		decfloat34(34)	calculatedunit	
CalculatedUnit	dd_cds_calculated_unit	char(50)		DD: CDS Calculated Unit

Figure 2.2 Technical Field Information of the CDS View from Listing 2.50

2.18 Conversion Functions for Currencies and Quantity Units

Conversion functions for the conversion of currencies and units of measure are based on several sets of persistent data records. These data records are included in the evaluation logic during processing of the conversion logic. The corresponding persistency captures, for example, the time-dependent conversion factors between currencies of different countries. This information must be integrated into the conversion logic based on the relevant key date.

Conversion functions in CDS views

The CDS syntax allows you to embed conversion functions directly into the implementation of your CDS views.

Example: Unit conversion

Listing 2.51 depicts such an embodiment for a unit conversion.

```
define view entity Z_ViewWithUnitConversions
  with parameters
    P_DisplayUnit : msehi
  as select from ZI_SalesOrderItem
{
  key SalesOrder,
  key SalesOrderItem,
      @Semantics.quantity.unitOfMeasure: 'OrderQuantityUnit'
      OrderQuantity,
      OrderQuantityUnit,
      @Semantics.quantity.unitOfMeasure: 'OrderQuantityDisplayUnit'
      unit_conversion( quantity        => OrderQuantity,
                       source_unit     => OrderQuantityUnit,
                       target_unit     => $parameters.P_DisplayUnit,
                       error_handling => 'FAIL_ON_ERROR' )
        as OrderQuantityInDisplayUnit,
      $parameters.P_DisplayUnit as OrderQuantityDisplayUnit
}
```

Listing 2.51 CDS View with Unit Conversion

Example: Currency conversion

Listing 2.52 shows a corresponding example of a currency conversion.

```
define view entity Z_ViewWithCurrencyConversions
  with parameters
    P_DisplayCurrency  : waers_curc,
    P_ExchangeRateDate : sydatum
  as select from ZI_SalesOrderItem
{
  key SalesOrder,
```

```
    key SalesOrderItem,
        @Semantics.amount.currencyCode: 'TransactionCurrency'
        NetAmount,
        TransactionCurrency,
        @Semantics.amount.currencyCode: 'DisplayCurrency'
        currency_conversion(
                  amount             => NetAmount,
                  source_currency    => TransactionCurrency,
                  target_currency    => $parameters.P_DisplayCurrency,
                  exchange_rate_date => $parameters.P_ExchangeRateDate,
                  exchange_rate_type => 'M',
                  round              => 'X',
                  decimal_shift      => 'X',
                  decimal_shift_back => 'X',
                  error_handling     => 'SET_TO_NULL' )
                as NetAmountInDisplayCurrency,
        $parameters.P_DisplayCurrency as DisplayCurrency
}
```

Listing 2.52 CDS View with Currency Conversion

Analysis of the examples

The conversion functions have mandatory input parameters as well as optional parameters. The latter only need to be supplied if you want to override the default values that are assigned to them automatically. Depending on the concrete input parameter of the chosen conversion function, you can supply it with a literal value, with an actual value from the respective data record, or with a parameter value.

Unit conversion

The unit conversion uses function `unit_conversion`. Earlier in Listing 2.47, the quantity (`quantity`) and its unit (`source_unit`) are provided as input parameters to the conversion function. These input parameters are bound to fields `OrderQuantity` and `OrderQuantityUnit` of CDS view `Z_ViewWithUnitConversions`. The semantic correlation between these two fields is expressed by annotation `@Semantics.quantity.unitOfMeasure:'OrderQuantityUnit'` of field `OrderQuantity`.

The target unit of conversion `target_unit` is defined by CDS parameter `P_DisplayUnit`. The result of the conversion is returned via field `OrderQuantityInDisplayUnit`. This field is related to unit field `OrderQuantityDisplayUnit`, which is derived from CDS parameter `P_DisplayUnit` via its annotation `@Semantics.quantity.unitOfMeasure: 'OrderQuantityDisplayUnit'`.

Currency conversion

The currency conversion (`currency_conversion`) from Listing 2.48 essentially corresponds to the previously discussed unit conversion. However, more input parameters are included in the calculation. Besides the amount

(`amount`), which is filled with the actual value of field `NetAmount`, the currency (`source_currency`), which is filled with the actual value of field `TransactionCurrency`, and target currency (`target_currency`), which is specified by CDS parameter `P_DisplayCurrency`, in particular, the key date of conversion `exchange_rate_date` must be specified by the function's caller. In our case, this is specified by CDS parameter `P_ExchangeRateDate`. Furthermore, the exchange rate type (`exchange_rate_type`) is set to fixed value `M`.

The currency conversion function allows you to activate or deactivate the business rounding (`round`) and the usage of shifts of decimal places before (`decimal_shift`) and after the calculation (`decimal_shift_back`). In the example described here, all these parameters are filled with constant literal value `X`. This valuation corresponds to the default values of the supplied parameters.

The relationships between the amount fields and their currency fields are expressed by annotations `@Semantics.amount.currencyCode:...`.

Error handling

Particular attention must be paid to the handling of possible errors when dealing with conversion functions. The two examples just described apply error handling `FAIL_ON_ERROR`. This corresponds to the default value of the corresponding input parameter and causes a runtime error in the database processing if the conversion can't be carried out successfully.

There are many possible root causes for such errors. For example, the unit conversion can't be executed successfully if the value of the target unit, which is supplied by the related input parameter, doesn't exist. Error handling `FAIL_ON_ERROR` therefore requires a high degree of data consistency and data completeness, as well as its own prevalidations of the supplied input parameters, for keeping possible error situations to a minimum.

Besides error-handling value `FAIL_ON_ERROR`, the conversion functions also support error-handling values `KEEP_UNCONVERTED` and `SET_TO_NULL`. Value `KEEP_UNCONVERTED` can be used for preserving the original value as the target value if problems occur in the context of the conversion. Value `SET_TO_NULL` can be used for setting the target value to `null` in case of issues. If you require any of these error-handling strategies, you should define the corresponding input parameter of the conversion function accordingly. However, note that you might not be able to detect issues easily if you choose an error-handling strategy that is different from `FAIL_ON_ERROR`.

> **Conversion Functions in Analytical Queries**
>
> In analytical query CDS views, the conversion functions for currencies and quantities are performed by the analytic engine after the records are aggregated. This can significantly improve the performance compared to

applying the same conversion functions already in the underlying data sources.

In addition, the analytic engine implements its own handling for conversion errors: when applying the conversion in an analytical query view, the original value with its unit is preserved if the requested conversion fails.

You'll find more information about analytical queries and the analytical engine in Chapter 10, Section 10.3.

2.19 Provider Contracts

Background information

CDS provider contracts formally define rules that the definition of a CDS model, which is classified accordingly, must follow. The rules consider predefined usage scenarios of the CDS models. Content-wise, the provider contracts correspond to the modeling patterns captured by annotation `@ObjectModel.modelingPattern`. Within a composition hierarchy of CDS models, only the root CDS model may be assigned a provider contract. Compositional children inherit the provider contract from their root.

The underlying rules of the provider contracts will ensure that CDS entities can actually be used as specified. They may result in both restricting the admissible modeling options and enabling framework-specific functions. Major parts of the rules are enforced by the CDS syntax check and as such may impact the activation of the CDS models, which means if you assign a provider contract to a CDS model but contradict its rules, it may not be possible to activate the CDS model. In contrast to the provider contract being defined by the corresponding CDS syntax elements, the aforementioned annotation `@ObjectModel.modelingPattern` doesn't influence the activation but serves documentation purposes only.

Overview

Table 2.8 presents on overview of the provider contracts and their intended usages.

Provider Contract	Usage
`transactional_interface`	This CDS projection view serves as a programmatic and modeled interface of the functionality of business objects in the ABAP RESTful application programming model context. Only simple projections of the underlying entities are supported.

Table 2.8 Provider Contracts

Provider Contract	Usage
`transactional_query`	This CDS projection view defines a service interface on top of a CDS view model for transactional processing. CDS projection views with this provider contract may be defined on top of regular CDS views without a provider contract or on top of CDS projection views with provider contract `transactional_interface`.
`analytical_query`	This transient query CDS projection view is executed by the analytical infrastructure.

Table 2.8 Provider Contracts (Cont.)

Example: CDS views with provider contracts

Listing 2.53 to Listing 2.55 demonstrate an example for the usage of the provider contracts in a CDS view stack. Fundamental CDS view `Z_ViewWithProviderContractA` from Listing 2.53 acts as the root CDS view of a compositional hierarchy.

```
define root view entity Z_ViewWithProviderContractA
  as select distinct from t000
{
  key abap.char'A' as KeyField
}
```

Listing 2.53 CDS View Z_ViewWithProviderContractA

CDS view `Z_ViewWithProviderContractB` from Listing 2.54 leverages CDS view `Z_ViewWithProviderContractA` as its data source. It's defined as a root CDS projection view with provider contract `transactional_interface` via the corresponding syntax element. In addition, this CDS view is annotated with `@ObjectModel.modelingPattern: #TRANSACTIONAL_INTERFACE`.

```
@ObjectModel.modelingPattern: #TRANSACTIONAL_INTERFACE
define root view entity Z_ViewWithProviderContractB
  provider contract transactional_interface
  as projection on Z_ViewWithProviderContractA
{
  key KeyField
}
```

Listing 2.54 CDS Projection View Z_ViewWithProviderContractB

CDS view `Z_ViewWithProviderContractB` itself is used as a data source of the transactional query CDS view `Z_ViewWithProviderContractC` from Listing

2.55, which is assigned CDS provider contract `transactional_query` and is annotated accordingly too.

```
@ObjectModel.modelingPattern: #TRANSACTIONAL_QUERY
define root view entity Z_ViewWithProviderContractC
  provider contract transactional_query
  as projection on Z_ViewWithProviderContractB
{
  key KeyField
}
```

Listing 2.55 CDS Projection View Z_ViewWithProviderContractC

2.20 Entity Buffer Definitions

Use cases

Entity buffer definitions allow you to speed up selections from CDS views in ABAP. This is achieved by processing selection requests directly on the ABAP application server instead of delegating them to the SAP HANA database.

Bufferable CDS views

To be able to buffer a CDS view, the implemented logic of the CDS view is subject to several restrictions. For example, CDS views with parameters and CDS views that use functions with varying results such as `utcl_current` can't be buffered. Those responsible for the CDS views need to document by means of annotation `@AbapCatalog.entityBuffer.definitionAllowed:true` that they are aware of these technical limitations and that they will consider these restrictions during the further lifecycle of the annotated CDS views.

Example: CDS view with buffer option

Listing 2.56 shows CDS view Z_ViewWithBufferA, which is annotated accordingly. It selects directly from table `T000`.

```
@AbapCatalog.entityBuffer.definitionAllowed: true
define view entity Z_ViewWithBufferA as select from t000
{
  key mandt as KeyField,
      mtext as Field
}
```

Listing 2.56 CDS View Z_ViewWithBufferA with Buffer Option

Buffering of stacked CDS views

If you want to enable a buffer for a CDS view, which uses another CDS view as its data source, not only the CDS view itself but also all the underlying CDS views must support buffers in principle.

Example: CDS view hierarchy with buffer option

CDS view `Z_ViewWithBufferB` from Listing 2.57 also supports buffering. This is technically feasible because its own data source—CDS view `Z_ViewWithBufferA`—also offers a buffer option.

```
@AbapCatalog.entityBuffer.definitionAllowed: true
define view entity Z_ViewWithBufferB as select from Z_ViewWithBufferA
{
  key KeyField,
      Field
}
```

Listing 2.57 CDS View Z_ViewWithBufferB with Buffer Option

Create entity buffer definitions

We now want to create an entity buffer definition. Entity buffer definitions can be created from within the ADT **Project Explorer** view. Therein, select the CDS view to be buffered and call function **New Entity Buffer** from its context menu. Enter the requested information in the dialog that is launched and confirm the creation.

Example: Entity buffer definition

In our case, we want to create a buffer entity definition for CDS view `Z_ViewWithBufferA` from Listing 2.56 that is named equally. This entity buffer definition will be assigned to the lowest definition `layer core` and specify a buffer strategy on a single record level (`type single`). Listing 2.58 demonstrates the corresponding definition.

```
define view entity buffer on Z_ViewWithBufferA
  layer core
  type single
```

Listing 2.58 Entity Buffer Definition for CDS View Z_ViewWithBufferA in the Layer Core

Layering of entity buffer definitions

There are multiple logical layers to which buffer definitions can be assigned. This allows specifying multiple entity buffer definitions for a single CDS view. However, in a single layer, there may at maximum be a single entity buffer definition per CDS view. The entity buffer definition of the highest layer determines the effective buffer handling.

Example: Overruling buffer definitions

Imagine that CDS view `Z_ViewWithBufferA` from Listing 2.56 and its buffer definition from Listing 2.58 could not be redefined (e.g., because they were shipped by SAP), but you would like to disable the buffer. You can achieve this by creating your own entity buffer definition for CDS view `Z_ViewWithBufferA` in a higher layer and deactivate the standard buffering from there. Listing 2.59 illustrates the corresponding entity buffer definition `ZZ1_ViewWithBufferA`.

```
define view entity buffer on Z_ViewWithBufferA
  layer customer
  type none
```

Listing 2.59 Entity Buffer Definition for CDS View Z_ViewWithBufferA in Layer Customer

It's assigned the highest `layer customer` and specifies via the declaration `type none` that no buffering will be applied. This setting overrules the entity buffer definition from Listing 2.58.

Buffer Handling

The definition and application of buffers will always be aligned with the actual data selections: on one side, buffers may restrict the further evolution of the buffered CDS views, and on the other side, you have to be aware that even if a buffer exists, not all selections can be executed on the ABAP application server. Furthermore, the administration of the buffers can imply a significant overhead and negatively impact resource consumption and performance. This is especially true if a huge amount of data is buffered, if the control settings of the buffers aren't chosen in an optimal way, or if the buffered data is invalidated frequently. You should also consider that entity buffers may be built up in parallel to existing table buffers. In such cases, you should evaluate whether both buffers are actually required or whether you could switch your selections for leveraging a single buffer only.

2.21 Summary

In this chapter, you learned how data provisioning can be implemented by leveraging SQL concepts such as joins, unions, and aggregation functions, as well as specialized CDS functions such as entity buffers. You also were introduced to CDS parameters. CDS parameters can be applied for enforcing user input as well as for providing context information that can be used for implementing selection logic.

In the next chapter, you'll learn more about associations, which were only briefly introduced in this chapter.

Contents

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

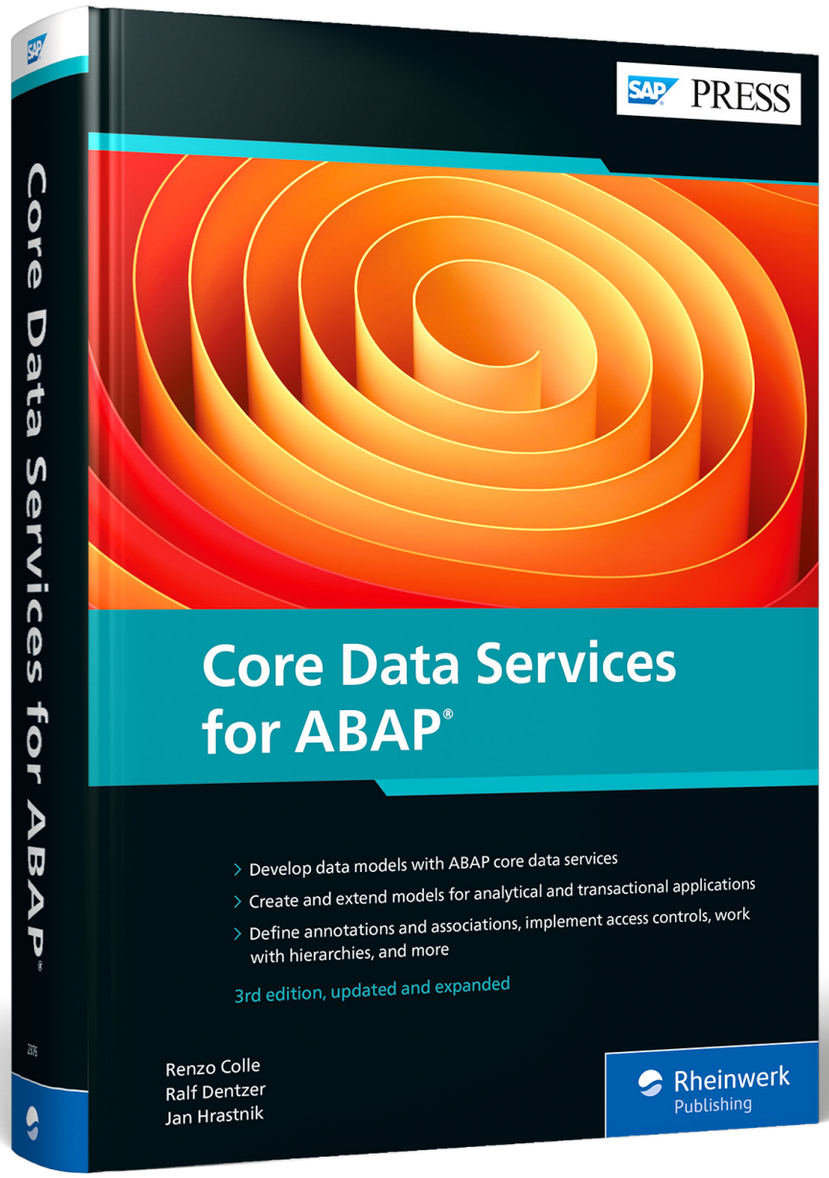
S

T

U

V

W



Renzo Colle is currently responsible for the end-to-end programming model of SAP S/4HANA in the central architecture group. He studied business mathematics at the Karlsruhe Institute of Technology (https://www.kit.edu/english) and has worked at SAP for more than 25 years in a wide variety of areas and roles. He started his career at SAP as a developer in strategic customer development. In SAP Business ByDesign, he was responsible for logistics and lead architect of the SAP ByDesign platform for cloud applications. As the inventor of the Business Object Processing Framework (BOPF) and lead architect of the ABAP RESTful application programming model, he has worked on model-driven software development and transactional applications for more than 20 years. Further information about Renzo can be found at https://de.linkedin.com/in/renzo-colle-30804ba1/de.

Ralf Dentzer has been working for several years in the central architecture group of the SAP S/4HANA suite with a focus on the use of core data services (CDS) in SAP S/4HANA. He joined SAP more than 25 years ago. He developed HR applications for SAP R/3, SAP ERP, and SAP Business ByDesign. After that, his tasks shifted to questions of overall architecture for new solutions. Ralf studied mathematics and received his doctorate from the University of Heidelberg. He is married and has two adult sons.

Jan Hrastnik is a member of the SAP S/4HANA cross architecture team, where he focuses on the virtual data model (VDM) and the use of CDS in ABAP applications. He has worked in various SAP development areas for more than 20 years. At the beginning of his career, he supported numerous customer projects in the automotive industry. Subsequently, he worked in the supply chain management development of SAP Business ByDesign. Jan's work initially focused on developing the master data required for the production processes before he took on overarching expert tasks for central architecture topics. He then worked on the SAP SuccessFactors Employee Central solution and native SAP HANA application development.

Colle, Dentzer, Hrastnik

Core Data Services for ABAP

754 pages | 10/2023 | $89.95 | ISBN 978-1-4932-2376-3

www.sap-press.com/5642